Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

M. Tiller
Ford Motor Company, USA
Implementation of a Generic Data Retrieval API for Modelica
pp. 593-602

Paper presented at the 4th International Modelica Conference, March 7-8, 2005,
Hamburg University of Technology, Hamburg-Harburg, Germany,
organized by The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/events/Conference2005/

Implementation of a Generic Data Retrieval API for Modelica

Dr. Michael M. Tiller
Ford Motor Company, Research and Advanced Engineering
Dearborn, MI USA

Abstract

For model developers, the Modelica modeling language is a valuable tool for describing the behavior of dynamic systems. However, developing models and performing analyses as part of a large scale engineering operation involves much more than creating behavioral descriptions [1],[2]. In order to integrate modeling and simulation into a typical product development process it is necessary to extract data (*e.g.* product information, part geometry, controller calibrations) from external sources.

This paper will describe an application programmer interface (API) for data retrieval that has been developed using the standard external function interface in Modelica. The API is composed of generic functions that can be implemented to extract data from a variety of external data sources. Such an API can be used to access data for material properties, part geometries, data tables, *etc*.

While the interface definitions are generic, our implementation of the generic API was specifically developed to retrieve data stored in XML [3] and utilizes the `libxml2` library [4] to retrieve and parse XML files containing product information. Furthermore, the API queries are performed using XPath expressions [5].

Currently, there is no standard API to allow Modelica models to retrieve information from external data sources. Hopefully this paper can demonstrate the power of such capabilities and prompt further discussion on formalizing a standard API with similar functionality.

Keywords: XML, XPath, HDF, MATLAB, Java

1 Introduction

Data is an integral part of modeling. Because Modelica is so often used for physical, first-principles modeling, there is typically a need to provide design data for numerous individual components. Such data is often available "somewhere" (we will use the term *external data source* as a generic term for sources of such data) but it must be collected to populate the Modelica model.

Because there is no standard way in Modelica to access such external data sources, this data is typically either entered by hand for each component or aggregated and organized into Modelica record definitions. We will refer to data managed in this way as a *Modelica representation* of the data.

While Modelica representations can be used, there are numerous drawbacks when trying to integrate the resulting models into large scale engineering and analysis processes. For example, such data often already exists in an external data source and copying it into a Modelica representation is both tedious, redundant and error prone. It also makes models that depend on the Modelica representation of the data difficult to update as new data becomes available. The best approach is one that retrieves the data as needed from the centralized external data sources. For example, if product information is stored in a relational database somewhere within a company, the ideal situation would be that the information could be automatically extracted directly from that database.

Another problem with Modelica representations of the data is cataloging large collections of component data. Representing such data in Modelica means, in practice, that large datasets are loaded into the modeling environment when only a very small percentage of that data is used. For example, we have data for a large number of production engines. The space required to store the data for each engine is considerable. We currently store all this information in a hierarchy of Modelica records. Storing the data in this way means slower loading times and higher memory consumption even though any given analysis only requires the data for one particular engine. Another issue with cataloging the data is querying the data set to see what information is available. While most data management systems include formalized query systems, there is no functional

equivalent in Modelica to query languages such as SQL, XPath or XQuery [6].

Often times, different characterization data is needed for the same component depending on the desired level of fidelity. Representation in Modelica often results in a variety of record definitions associated with a given physical component (*i.e.* one for each level of fidelity). Typically these record definitions include large amounts of redundant data between them. However, because of the semantics associated with records[1] in Modelica, it is difficult to eliminate such redundancy.

Use of data expressed directly in Modelica typically results in that data being "hard-wired" into the resulting simulation. Although the data can be changed it is typically a manual process and impractical for large data sets. Ideally it should be possible to load the data on demand from a data source in the event that such a data source has been changed or updated.

Another concern is storage of the data. A centralized data source is often accessed over the network. As such, the data is only stored in one place. This not only conserves space but also provides a definitive source for the data. If data is represented in Modelica there is the risk that variations will develop across multiple copies of the data. Loading data on demand over a network provides a more dynamic system for data management.

Some applications require very large data sets to be available but only use relatively small chunks at any given time [7]. In such a case, a system that is able to load data into memory for use by a model on an "as needed basis" can save a considerable amount of space (*e.g.* in results files). By avoiding the need to represent the entire dataset in Modelica and the compilation process (*e.g.* symbolic analysis) avoids the need to read in and analyze such data.

2 Interface

For these reasons, we have developed an API that allows us to retrieve data from external sources. This is not a "database API" because it does not include the complete set of operations typically associated with database interactions (*e.g.* changing data, committing transactions, *etc*). Instead, the focus for this package is on retrieval only. The interface is generic so it could be mapped to a wide variety of external data sources (including, but not limited to, databases).

In this section we will go through the API in detail to explain the basic functionality before moving on to a discussion of our implementation of the interface and some examples of its use.

The data retrieval API is implemented within a package called `DataRetrieval`. The package contains several class definitions that extend from the ExternalObject class used for handling opaque references to external (*e.g.* C language) data. In addition, it contains several functions that operate on these locally defined data types.

2.1 Opening and Closing a Data Source

In order to access a data source it is first necessary to open it for queries by instantiating an object to represent the data source. This object can then be used in subsequent query operations. To open a database, a `Source` object must be created, *e.g.*

```
import DataRetrieval.*;
parameter Source s=Source(
                        format="…",
                        url="…",
                        context="…");
```

where `format` identifies the format of the data source (*e.g.* "XML"), `url` is a string encoded using the *uniform resource locator* (URL) syntax [8] and `context` is used, in a data source specific way, to limit the scope of subsequent queries.

The ExternalObject interface also provides for a destructor although that is not called directly so the details are not included.

2.2 Query Expressions

Once a data source is available (in the form of a `Source` object), queries can be made against it. Because Modelica is a strongly typed language and it is currently not allowed to overload functions, query functions are defined for specific data types (*i.e.* String, Real, Integer and Boolean) and for specific dimensionalities (*e.g.* scalars, vectors, matrices, *etc*). But, each query function relies on a common query expression syntax.

The precise semantics of the query expressions do not necessarily have to be defined for each data source. The generic aspect of the API does not interact in anyway with the semantics of these expressions. As we will discuss shortly, our implementation uses XPath expressions for such queries.

[1] Specifically, the strict requirement that assignment is only possible between identical record types.

related information and that information for different levels of model fidelity can be grouped together in the same data source.

2.6 Querying Available Choices

Our data retrieval API is built around the idea of query expressions. In most of the previous sections it is assumed that the query expression is written to match exactly one piece of data. However, allowing query expressions to match multiple pieces of data can be quite useful because it would provide tools with the ability to identify all data that is potentially compatible for a specific data type. For example, when loading records that characterize electric motors it is useful to query a data source for compatible data and use it in the same way that the `choices` family of annotations are used.

The current version of our API does not provide such functionality for two reasons. First, such functionality would require tool support. The other reason is that such functionality would require certain concepts (*e.g.* ordering, filtering, *etc*) not current expressed in the data retrieval API. In Section 6 we will discuss how such capabilities could be implemented with some degree of tool support and a slightly more sophisticated querying scheme.

3 Implementation

Up to this point, the discussion has been completely generic with only a few fragments of actual code and only vague discussions on query expressions. In the next two sections we will describe an implementation of the API and get into specific detail about how it can be used.

Our implementation was developed specifically to extract data from XML documents. Such documents may exist on web servers or they may be stored in local files. XML is fast becoming an important technology in all aspects of computing because of its ability to structure information in an operating system, programming language and application neutral way. In addition to existing high quality implementations [9], there are several advances on the horizon that will support handling of large collections of binary data [10], [11] (*e.g.* simulation results).

We treat each XML document as an object-oriented database (OODB). An OODB is useful for storing heterogeneous collections of objects. In our experience, engineering data (part dimensions, test data, *etc*) fits quite well into OODBs.

But storing the data is only one aspect that we need to worry about. The other aspect is querying our data source to extract data. For this, our implementation uses XPath, a standard for "addressing parts of an XML document". XPath provides a standardized way of identifying what data in an XML document we wish to extract. A similar emerging standard is XQuery [6] which may prove to be a superior (and mostly backwardly compatible) technology once it is formally standardized.

Consider the sample engineering database shown in Figure 1. We will use this trivial database to demonstrate the capabilities of the XPath standard. Although the data and structure of the database are quite simple, all these examples could be applied to much larger databases without alteration.

```
<?xml version="1.0"?>
<engineering_data>
  <product name="ZY300">
    <base_motor type="C12"/>
  </product>
  <part>
    <motor name="C12">
      <rotorJ>0.011</rotorJ>
    </motor>
  </part>
</engineering_data>
```

Figure 1: Sample Engineering Database

Let's begin with a simple example. Imagine we wanted to extract the name of the base motor used in the ZY300 product. Using the query expression:

```
//product[@name='ZY300']
 /base_motor
 /attribute::type
```

to query the database shown in Figure 1 would return '`C12`'. The '`//`' at the start of the request means "at any level in the document hierarchy". The '`product`' string following this is interpreted as the name of the element type that is being requested. Anything contained in '`[]`'s represents a predicate. Elements for which the predicate is false will be filtered. In this predicate '`@name`' represents the attribute '`name`'. So the first line locates the ZY300 product in the database. Each subsequent '`/`' in the

expression is used to indicate traversal one level deeper into the hierarchy. If a name is prefixed by 'attribute::' that indicates that the query is for an attribute rather than an element. So the complete expression can be interpreted as "Search the hierarchy for product elements whose name attribute is ZY300 and for each of these find the base_motor element immediate below it and return the value of the type attribute for that element".

Imagine we wish to extract the value contained between the rotorJ tags in Figure 1. We can extract that data with the following XPath expression:

```
//motor[@name='C12']
 /rotorJ/text()
```

This query is quite similar to the previous query except it uses the 'text()' function to return the textual content within the rotorJ element.

But now let's look at a more challenging example. In the second example, we assumed that we knew the model name for our motor, C12, *a priori*. Imagine we want to extract the rotational inertia of the rotor but we don't know the motor name. Instead, what we know is that it is the base motor used in the ZY300 product? In this case, we can combine the two queries we made previously into:

```
//motor[@name=
  //product[@name='ZY300']
   /base_motor
   /attribute::type]
 /rotorJ/text()
```

With this example we have nested our requests for the type of base motor used inside a predicate used to search for the motor. By using the query for the ZY300 base motor type in the predicate involving the motor name, we were able to identify rotor inertia based on its relationship to the ZY300 product rather than by name.

These are a few examples of the kinds of queries that are possible with XPath expressions. This is by no means a complete introduction to XPath. Instead the goal of these examples was to provide sample expressions so that expressions in subsequent examples can be interpreted.

It is important to note that parsing XML, converting it into a traversable data structure and implementing an XPath query engine are not trivial tasks. Fortunately, there are multiple implementations of these standards that can be used as off-the-shelf software components. For our work, we chose to use the libxml2 [4] library that was developed for use with the Gnome desktop environment. The libxml2 library includes complete, robust implementations of many XML related standards including DOM [12], SAX, XPointer [13] and XPath [5].

4 Examples

4.1 Retrieving Parameter Values

One of the most common uses of the data retrieval API is to supply parameter values in a model. In this section we will show how the data retrieval API can be combined with a sample data set (shown in Figure 2) and specific query expressions (using the XPath notation described in Section 3) to accomplish this task.

```
<?xml version="1.0"?>
<engine_data>
  <engine name="Beta">
    <real name="bore">88.2</real>
    <stroke>84.0</stroke>
    <val name="conrod" units="mm"
         value="125.0"/>
  </engine>
  <engine name="Gamma">
    <real name="bore">87.2</real>
    <stroke>85.0</stroke>
    <val name="conrod" units="mm"
         value="123.7"/>
  </engine>
</engine_data>
```

Figure 2: Sample Engine Data

An important thing to note about the engine data shown in Figure 2 is that each parameter (bore, stroke and conrod) are represented using different XML constructs. The engine bore is represented as the text inside a generic real element, the stroke appears as the text inside a special element type of its own and the connecting rod length is given by an attribute associated with another generic element type, val, but with a specific string, conrod, given for its name attribute.

So the challenge in this example is to show how XPath syntax is expressive enough to allow us to address each piece of data even though the contexts are quite different. Figure 3 shows the various XPath expressions that can be used to extract the necessary data from the XML file.

```
model TestReals
  import DataRetrieval.*;
  parameter Source engine =
    Source(format="XML",
           url="engines.xml");
  parameter Real bore =
    getReal(source=engine,
      name="//engine[@name='Beta']
            /real[@name='bore']
            /text()");
  parameter Real stroke =
    getReal(source=engine,
      name="//engine[@name='Beta']
            /stroke/text()");
  parameter Real conrod =
    getReal(source=engine,
      name="//engine[@name='Beta']
            /val[@name='conrod']
            /attribute::value");
end TestReals;
```

Figure 3: Parameter Extraction

4.2 Populating Records

The example shown in Figure 3 includes several complex XPath expressions. Because it can be difficult to formulate such expressions and because entering them manually or copying and pasting them several times can be error prone and/or difficult to maintain, it is desirable to try and encapsulate these expressions somehow. One way to accomplish this in Modelica is to create a special record type for the data and then create a function that can populate such a record automatically. For example, consider the following Modelica record definition:

```
record EngineData
  import Modelica.SIunits.*;
  parameter Diameter bore;
  parameter Length stroke;
  parameter Length conrod;
end EngineData;
```

To populate such a record with data from the file shown in Figure 2, we could write a function that constructed such a record from the name of the engine and the location of the data. Figure 4 shows what such a function might look like.

```
function Load
  import DataRetrieval.*;
  input String engine;
  input String url;
  output EngineData data;
protected
  String context=
   "//engine[@name='"+engine+"']";
  Source source=
    Source(format="XML", url=url,
           context=context);
algorithm
  data.bore :=
    getReal(source=source,
      name="real[@name='bore']
            /text()");
  data.stroke :=
    getReal(source=source,
      name="stroke/text()");
  data.conrod :=
    getReal(source=source,
      name="val[@name='conrod']
            /attribute::value");
end Load;
```

Figure 4: Populating a record

The function shown in Figure 4 also highlights another feature of the data retrieval API. When the `source` object is created, the optional `context` argument is used to define the context in which all subsequent XPath expressions should be evaluated. What this means in practice is that it is assumed that any queries associated with the `source` object apply only to the specific engine for which data is being retrieved. In this way, the query expressions for each invocation of `getReal` can leave off the engine selection prefix expressions resulting in shorter path expressions.

Once defined, the `Load` function shown in Figure 4 can then be used to provide values for a record without any need to include calls to the data retrieval API or any XPath expressions, e.g.

```
parameter EngineData engine =
  Load(engine="Gamma",
  url="engines.xml");
```

4.3 Loading Arrays

While loading a complete array into Modelica for use in a model is an obvious example of how the data retrieval API might be used, there are also other reasons why you might want to load only a partial array. Consider the case of cubic interpolation. Imagine we have interpolation data that is stored in an array as follows:

$$\begin{bmatrix} x_1 & f(x_1) & f'(x_1) \\ \cdots & \cdots & \cdots \\ x_n & f(x_n) & f'(x_n) \end{bmatrix}$$

Now, if we need to construct the cubic polynomial approximation for any value x, we only need to know the values for the function and its derivative associated with x_i and x_{i+1} (where $x_i <= x <= x_{i+1}$). The important point is that we do not need to load the entire matrix into a Modelica variable. Instead, we could simply extract the values that we need at any given time and construct the approximations in a piecewise form. So given the following data file:

```
<?xml version="1.0"?>
<data>
  <function name="z">
    <point x="0" f="0" df="0"/>
    <point x="1" f="0" df="1"/>
    <point x="2" f="1" df="0"/>
    <point x="3" f="0" df="-1"/>
    <point x="4" f="0" df="0"/>
  </function>
</data>
```

We can use the following Modelica code to evaluate the function "z" described in the data file:

```
  parameter RealMatrix data =
    RealMatrix(source=f,
      rows="//function[@name='z']
            /point",
      cols="attribute::x
           |attribute::f
           |attribute::df");
  Interpolate2 y(x=time,
                 data=data);
```

This code defines the contents of the matrix using the XPath expressions and then passes it to a model which only extracts the function and derivative values for the two closest points at any given time. Now, formulating a cubic polynomial approximation for a simple 1D function does not necessarily require such powerful functionality. However, if we wanted to construct a 3D approximation for a relatively large data set [7] using a complex cubic interpolation scheme this API could help us minimize memory consumption while still exposing the underlying mathematical structure.

4.4 Generating "choices"

As mentioned previously, an XPath expression might match several different entities in an XML document. For example, if we wanted to extract the names of all engines present in Figure 2, we could express this with the XPath expression:

```
//engine/attribute::name
```

The results of such a query could then be used in subsequent queries to select from elements in an XML document. As mentioned previously, this capability would require some degree of tool support.

5 Discussion

5.1 Alternative Source

While the data retrieval API is generic, the implementation discussed in this paper assumes that the data will be represented natively in XML and the query expressions will follow the XPath specification. But there are several other formats that are frequently used to store data and for which a retrieval API might be useful. Examples of these would include HDF [14] and the MATLAB ".mat" file formats [15].

The only significant impact of changing the format of the underlying data source is on the query expressions. There are two ways to approach query expressions in such cases. First, for each format a (potentially) unique query expression syntax could be used. This would allow, for example, SQL to be used if the underlying data source was a relational database. The drawback of this approach is that it would be impossible to write general functions (*e.g.* the Load function for loading engine data shown in Figure 4) for an arbitrary data source. Instead, a function would have to be defined for each potential data source format.

On the other hand, if each data source used the XPath approach for querying, then a consistent syntax would be available across the various platforms. The advantage of this is that users would only need to be familiar with XPath and no other query expression format. The difficulty is that XPath applies to

XML, not to other formats. One way to bridge the gap would be to define a mapping from each format to XML. For example, consider the following MATLAB code which writes several matrices to a file:

```
>> A = [1, 2, 3; 4, 5, 6];
>> B = [6,7; 8,9; 10,11];
>> save 'AB.mat' -V4 A B
```

The contents of the file `'AB.mat'` are stored in the MATLAB specific format. But for the purposes of formulating queries we could create a mapping that defines a translation to XML that would result in an XML document that looks like this:

```
<?xml version="1.0">
<MATLAB>
  <matrix name="A">
    <row><col>1.0</col>…</row>
    <row><col>4.0</col>…</row>
  </matrix>
  <matrix name="B">…</matrix>
</MATLAB>
```

In this way, it would then be possible to load data from MATLAB using the data retrieval API with code like

```
parameter Source f =
  Source(format="MAT4",
         url="AB.mat");
parameter RealMatrix data =
  RealMatrix(source=f,
    rows="//matrix [@name='A']
           /row ", cols="col");
```

Note that the data itself would not necessarily have to be translated into XML. Instead, a special XPath interpreter could be developed for each format that understood the "mapping" involved.

5.2 Data Management

The goal of this API is not just to provide a package for opening and querying data sources. In addition, the design goals are also meant to address nagging problems with handling data in Modelica. With this new API we can avoid loading large amounts of data either as constants or definitions in packages (*e.g.* Modelica.Media idea gas data) and we can avoid (through the selective extraction functions) loading entire data sets into Modelica variables when only a subset are needed at any given time during a simulation.

In addition, data stored in Modelica typically ends up being compiled into simulations. In a sense, the data is then frozen inside the analysis. Any change in the data requires the model to be recompiled or have its input files modified in some way. By relying on external data sources, the "fresh" data can be loaded on demand.

5.3 Modelica Deficiencies

While the external function interface in Modelica provided enough functionality to implement the API and create functioning examples, there are still a few areas where Modelica could be improved.

First, the API structure would benefit greatly from support for methods that can be invoked on user defined classes. Without methods, special functions much be written and type information about arguments and return types must somehow be aggregated to form unique function names. In addition, features to support better abstraction and polymorphism support would allow specialized `Source` objects to be developed (e.g. `XMLSource`, `HDFSource`) but remain compatible with all existing functions that required `Source` objects as arguments. As things stand currently, the definition for the `Source` class must be familiar with all potential formats (hence the `format` argument) but with the ability to subclass, new formats could be supported without the need to change or update the existing `Source` definitions.

Another issue with Modelica is units. While the language allows unit information to be associated with variables and data sources may include unit information, the current API specification does not exploit any of this information. Built-in unit conversion capabilities in Modelica might make it possible to handle units without having to implement any manual unit conversions.

Finally, the XML related tools used in this implementation were available in C and could be integrated nicely through the Modelica external function interface. However, more and more of these capabilities are appearing in Java. As things currently stand, it is not possible to leverage Java code through the external function interface although it would be nearly trivial to do so. By including an instance of a Java Virtual Machine in Modelica tools and/or generated code, it would be possible to easily load Java classes into memory and invoke functions (and perhaps methods) defined in Java. Simply de-

fining how arguments are passed to and from Java code would enable leveraging tremendous amounts of existing Java code.

5.4 DTDs and Namespaces

Two features of XML not discussed in this paper are Document Type Definitions (DTDs) and namespaces. This section includes some discussion about these topics and how they relate to our data retrieval API.

DTDs define a specific schema associated with an XML document. We could have forced all XML data to be used with the data retrieval API to follow a specified DTD. This would have made retrieval considerably simpler because we could have anticipated, to a greater degree, the structure of the data we were trying to retrieve. However, it is quite impractical to expect that external sources of data will always conform to a specified DTD. It is possible to translate such data from its native format into a form that conforms to a specific DTD but this would likely involve more work than our approach and would still involve XPath or something similar. The strength of our approach is the ability to use it in conjunction with arbitrarily structured data.

Namespaces could also be useful in annotating existing datasets with new element types that are explicitly tagged to be specially included for our purposes. In such a scenario, special tags could be defined within a namespace and then added to existing XML documents. These specialized elements should, in theory, be ignored by other applications since they belong to a namespace that the application is unfamiliar with. This would add a level of complexity to the implementation and the need to specially annotate external data sources but without any real benefit. For this reason, we did not utilize namespaces.

6 Benefits of Standardization

While we have created this implementation for our own purposes based on identified needs in our organization, it is quite likely that many Modelica users would benefit from a standard data retrieval API like the one described in this paper. In this section, we highlight some of the benefits a standard API would have over the "user space" implementation we have created.

First, query expressions could be used to generate lists of "choices" much like the existing `choices` annotation. Such functionality would have to be available (*i.e.* compiled into) Modelica tools in order to link such information to the graphical user interface. The current external function interface is, at least in the case of Dymola, limited to user simulations and such external functions are not available to the Dymola process itself.

Another advantage of a standard data retrieval API is that it could be used within the standard Modelica libraries to manage data. For example, the Modelica.Media library contains a tremendous amount of data associated with ideal gases. This data could be stored outside the Modelica environment and loaded selectively on an as-needed basis.

As mentioned previously, our API is implemented through the Modelica external function interface and, as such, is not available to the Dymola GUI. This makes model checking and model compilation impossible for cases where variables in Modelica are dimensioned based on calls to the API (*i.e.* to determine the full size of an external matrix). By standardizing the API, it would be possible to use external data to dimension variables used in Modelica.

7 Conclusions

In order to integrate Modelica models with existing engineering and analysis processes, retrieval of data from external data sources for use in models is essential. This paper outlines one way such integration can be accomplished. Use of XPath expressions is a powerful component of our implementation and, through formalized mappings as described in Section 5.1, this approach to querying can be extended to other non-XML based data sources as well. Our implementation focuses only on XML documents as data sources and represents only a proof-of-concept implementation (*e.g.* no caching is performed in our implementation).

Standardization of this API opens up many possibilities for integration of database information into graphical model development environments. It could also automate the tedious and error prone process of writing special functions (like the one shown in Figure 4) to populate records used to characterize models.

The topic of a formalized API for retrieving external data sources has come up occasionally in Modelica design meetings. Hopefully this implementation can serve as a starting point for further discussions, proposals and eventually standard functionality available to all Modelica users.

References

[1] Tiller M., Bowles P., and Dempsey M., "Development of a Vehicle Model Architecture in Modelica". *Proceedings of the 3rd International Modelica Conference*, Nov. 3-4, 2003. Linköping, Sweden.

[2] Tiller M, "Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications". *Proceedings of the 3rd International Modelica Conference*, Nov. 3-4, 2003. Linköping, Sweden.

[3] "Extensible Markup Language (XML)", http://www.w3.org/XML/

[4] "The XML C parser and toolkit of Gnome", http://www.xmlsoft.org/

[5] "XML Path Language (XPath) Version 1.0", http://www.w3.org/TR/xpath

[6] "XQuery 1.0, An XML Query Language", http://www.w3.org/TR/xquery/ (working draft)

[7] Newman C. E., Batteh J. J., and Tiller M., "Spark-Ignited-Engine Cycle Simulation in Modelica", *Proceedings of the 2nd International Modelica Conference*, March 18th-19th, 2002. Oberpfaffenhofen, Germany.

[8] Berners-Lee T., Fielding R., and Masinter L., "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396. http://www.ietf.org/rfc/rfc2396.txt

[9] "Berkeley DB XML 2.0", Sleepycat Software, http://www.sleepycat.com/products/xml.shtml

[10] "XBIS XML Information Set Encoding", http://xbis.sourceforge.net/

[11] "XML-binary Optimized Packaging", http://www.w3.org/TR/xop10/

[12] "Document Object Model (DOM)", http://www.w3.org/DOM/

[13] "XML Pointer Language (XPointer)", http://www.w3.org/TR/xptr/

[14] "HDF 4.1r3 User's Guide", http://hdf.ncsa.uiuc.edu/UG41r3_html/

[15] "MAT-File Format, Version 7", http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matfile_format.pdf